МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Жадный алгоритм и A*

Студент гр. 0304 _____ Алексеев Р.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2022

Цель работы.

Изучить жадный алгоритм и алгоритм A* для нахождения путей в графе.

Реализовать жадный алгоритм и алгоритм A*.

Задание.

Вариант 2

В A* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Задание 1

Разработайте программу, которая решает задачу построения пути в *ориентированном* графе при помощи **жадного алгоритма**. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины

Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

abcde

Задание 2

Разработайте программу, которая решает задачу построения кратчайшего пути в *ориентированном* графе **методом A***. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Пример входных данных

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины

Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

ade

Выполнение работы.

Описание алгоритма

Жадный алгоритм

Жадный алгоритм находит путь из начальной вершины к конечной. Алгоритм заключается в принятии локально оптимальных решений на каждом шаге, допуская, что конечное решение также окажется оптимальным. Найденные алгоритмом путь не всегда является минимальным.

На каждом шаге алгоритма выбирается ребро с минимальным весом, исходящее из текущей вершины, и по нему выбирается новая вершина для следубщего шага. Шаги повторяются пока не будет достигнута конечная вершина.

Если на очередном шаге алгоритма из текущей вершины нет исходящих ребер по которым ещё не проходили, то переходим к предыдущей вершине и проверяем наличие исходящих ребер из неё.

Чтобы избежать зацикливания алгоритма, проход по любому ребру возможен только один раз.

A*

A* - алгоритм поиска наименьшего по стоимости пути от начальной вершины к конечной.

Алгоритм состоит из нескольких шагов:

1. Для каждой вершины-ребенка текущей вершины находится значение пути до неё от начальной вершины и значение эвристической функции.
2. В качестве новой текущей вершины выбирается вершина с наименьшим значением суммы пути и эвристической функции из всех вершины, в которые можно перейти из обработанных вершин и которые ещё не были обработаны.
3. Если на очередном шаге достигнута конечная вершина, значит путь найден, работа алгоритма прекращается.

Чтобы получить итоговый результат, необходимо пройти по найденному пути в обратном направлении, сохраняя пройденные вершины.

Сложность алгоритма

Жадный алгоритм

Жадный алгоритм на каждом шаге выбирает ребро с минимальным весом, чтобы перейти по нему в следующую врешину, чтобы выбрать минимальное ребро, необходимо отсортировать список ребер, поэтому сложность жадного алгоритма по числу операций — O($v\ log v$), по памяти — O(v), где v — число вершин в графе, т. к. необходимо хранить только список вершин и строку пути, которая не может превышать по количеству символов количества вершин в графе.

A*

Сложность алгоритма A* по числу операций зависит от эвристической функции. В худшем случае она будет экспоненциальной, но если эвристическая функция удовлетворяет условию:

$$|h(x)-h^*(x)|\leq O(\log(h^*(x)))$$

где h — эвристическая функция, h* - оптимальная эвристическая функция, т. е. точная оценка расстояния от вершины к цели.

Сложность по количеству операций для A* будет O(*B^D*), где *B* — среднее количество исходящих ребер, *D* — длина кратчайшего пути. Сложность по памяти будет O(*B^D*).

Если использовать оптимальную эвристическую функцию, то сложность по количеству операций и памяти станет полиномиальной. Количество рассмотренных узлов будет $N+1=1+B+B^2+...+B^D$, следовательно, при *B* = 1 количество рассмотренных вершин будет равняться *D*, т. е. сложность по памяти — O(*D*), по числу операций — O(*D*).

Функции и структуры данных

Для хранеия информации о вершинах был создан класс *GrafVertex*.

Класс имеет четыре поля: *self.symbol* — символ, которым обозначена вершина, *self.parent* — индекс вершины, из которой перешли в эту в A*, *self.child* — список детей вершины, *self.heur* — эвристическая функция.

Для работы с вершинами реализован ряд методов.

Метод *setSymbol(self, symbol)* используется для установки символа вершины. Метод принимает: *symbol* — символ. Метод сохраняет полученный символ в поле *self.symbol.*

Метод *addPar(self, newPar)* используется для установки родителя вершины. Метод принимает: *newPar* — индекс родителя. Метод сохраняет полученный индекс в поле *self,parent*.

Метод *addChild(self, newChild)* используется для добавление нового ребенка. Метод принимает: *newChild* — новый ребенок. Метод добавляет нового ребенка в список *self.child*.

Для вывода вершины в консоль перегружен метод *__str__(self)*. Метод сохраняет в строку символ вершины, индекс родителя и список детей. Метод возвращает получившеюся строку.

Для обработки входных данных создан класс *Graf*.

Класс имеет шесть полей: *self.numStart* — индекс начальной вершины, *self.numFinish* — индекс конечной вершины, *self.graf* — список вершин графа, *self.result* — результат работы алгоритмов, *self.interConc* — флаг промежуточных выводов, *self.individual* — флаг индивидуализации.

Для обработки данных создан ряд методов.

Метод *printResult(self)* используется для вывода результата. Метод выводит содержимое поля *self.result* в консоль.

Метод *createGraf(self, edgesBuf)* используется для создания графа. Метод принимает: *edgesBuf* — список ребер с весами. Метод создает вершины графа, сохраняя в них информацию о символах и детях.

Метод *greedyAlg(self)* используется для реализации жадного алгоритма. Метод перебирает детей текущей вершины, первой текущей вершиной является начальная, и переходит к тому, вес ребра к которому меньше всего. Если найдена конечная вершина, то работа метода останавливается. Если из текущей вершины никуда нельзя перейти, то в качестве текущей вершины берется та, которая была последней добавлена в результат. Каждая пройденная вершина добавляется в результат.

Метод *algAStar(self)* используется для реализации алгоритма A*. Метод, начиная с начальной вершины, добавляет вершины в очередь с приоритетом, где в качестве приоритета служит сумма пути до вершины от начальной вершины и разница между кодами символов в таблице ASCII. Метод перебирает вершины очереди, извлекая вершину с наименьшей суммой, при попомщи вызова метода *getVer(self, queueVer, f)*, проверяет её детей, если их нет в очереди, добавляет их, иначе сравниват их значение

суммы стоимости пути и разницы кодов с новым, заменяя при необходимости. Если найдена конечная вершина, то метод вызывает метод *createResult(self)*. Если в очереди закончатся вершины, а конечная вершина не будет достигнута, значит пути до неё нет.

Метод *getVer(self, queueVer, f)* используется для получения вершины из очереди. Метод принимает: *queueVer* — список индексов вершин в очереди, *f* — список приоритетов вершин. Метод сравнивает приоритеты вершин из очереди и возвращает индекс вершины с минимальным приоритетом.

Метод *createResult(self)* используется для создания ответа после работы A*. Метод, начиная с конечной врешины, переходит по родителям вершин, пока не достигнет начальной вершины. Пройденные вершины сохраняются в поле *self.result*.

Для решения первой задачи создана функция *firstTask(interConc)*. Функция принимает: *interConc* — флаг промежуточных выводов. Функция ссчитывает ввод с клавиатуры начальной и конечной вершин, набора ребер с весами. После считывания входных данных функция создает экземпляр класса *Graf* и вызывает необходимые методы класса.

Для решения второй задачи создана функция *secondTask(interConc)*. Функция принимает: *interConc* — флаг промежуточных выводов. Функция ссчитывает ввод с клавиатуры начальной и конечной вершин, набора ребер с весами. После считывания входных данных функция создает экземпляр класса *Graf* и вызывает необходимые методы класса.

Для решения второй задачи с индивидуализацией создана функция *firstTask(interConc)*. Функция принимает: *interConc* — флаг промежуточных выводов. Функция ссчитывает ввод с клавиатуры начальной и конечной вершин, набора ребер с весами. После считывания входных данных функция создает экземпляр класса *Graf* и вызывает метод *createGraf(self, edgesBuf)*,

после чего считывает ввод значений жвристической функции для каждой вершины. После считывания функция вызывает необходимые методы класса.

Выводы.

Были изучены жадный алгоритм и A* для нахождения путей в графе.

Была написанна программа, находящая оптимальный путь при помощи жадного алгоритма и минимальный путь при помощи алгоритма A*. Программа была дополнена таким образом, чтобы для A* можно было ввести значения эврестической функции с клавиатуры.

Было установлено, что сложности жадного алгоритма по числу операций — O(v *logv*), где *v* — число вершин в графе, и по памяти — O(v). Сложности A* по числу операций и памяти в худшем случае — O(*B*^*D*), где *B* — среднее число исходящих ребер, *D* — длина кратчайшего пути, в лучшем случае — O(*D)*.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class GrafVertex:       # Вершина графа
    def __init__(self):
        self.symbol = None      # Символ вершины
        self.parent = None      # Индекс родителя, через которого
пришли
        self.child = []         # Список индексов потомков, весов
ребер к ним и флагов посещения
        self.heur = None        # Эвристическая функция

    def setSymbol(self, symbol):    # Установить символ вершины
        self.symbol = symbol

    def addPar(self, newPar):   # Добавить родителя
        self.parent = newPar

    def addChild(self, newChild):   # Добавить потомка
        self.child.append(newChild)

    def __str__(self):      # Вывод вершины
        string = f"Вершина: {self.symbol}\n Индексы родителей:\n"
        string += f"Родитель: {self.parent}\n"
        string += " Индексы потомков:\n"
        for i in self.child:
            string += f"  {i}\n"
        return string


class Graf:
    def __init__(self, start, finish, interConc, individual):
        self.numStart = ord(start) - ord('a')                   #
Индекс начальной вершины
        self.numFinish = ord(finish) - ord('a')                 #
Индекс конечной вершины
        self.graf = [GrafVertex() for _ in range(26)]   # Список
вершин
        self.result = ""                                        #
Результат

        self.interConc = interConc                              #
Промежуточные выводы
        self.individual = individual                            #
Индивидуализация

    def printResult(self):      # Вывод результатов
        if self.interConc:
            print("\n\033[34mРезультат:\033[0m")
        print(self.result)

    def createGraf(self, edgesBuf):     # Создание графа из ребер
```

```
            if self.interConc:
                print("\033[33mСоздание графа из ребер:\033[0m")
            for edge in edgesBuf:
                    numSVer = ord(edge[0]) - ord('a')       # Индекс
вершины, где 97 - номер символа 'a' в ASCII
                numFVer = ord(edge[1]) - ord('a')
                weight = float(edge[2])
                if self.interConc:
                                    print(f"\tДобавление  в  граф  ребра
{edge[0]}——{weight}—>{edge[1]}")
                 if not self.graf[numSVer].symbol:   # Если начальной
вершины ребра ещё нет
                    if self.interConc:
                            print(f"\tВершины \"{edge[0]}\" ещё нет в
графе. Добавляем её.")
                    self.graf[numSVer].setSymbol(edge[0])
                 if not self.graf[numFVer].symbol:   # Если конечной
вершины ребра ещё нет
                    if self.interConc:
                            print(f"\tВершины \"{edge[1]}\" ещё нет в
графе. Добавляем её.")
                    self.graf[numFVer].setSymbol(edge[1])
                self.graf[numSVer].addChild([numFVer, weight, True])
                if self.interConc:
                    print("")
            if self.interConc:
                print(f"\tГраф построен.\n")

        def greedyAlg(self):    # Жадный алгоритм
            if self.interConc:
                print("\033[32mЗапуск жадного алгоритма:\033[0m")
            numVer = self.numStart  # Номер начальной вершины
              self.result += chr(numVer + ord('a'))      # Добавляем
начальную вершину в результат
            while True:
                    if numVer == self.numFinish:     # Если текущая
вершина - финальная, то путь найден
                    if self.interConc:
                           print("\tДошли до конечной вершины, работа
алгоритма окончена.\n")
                    break

                bufWeight = 0
                bufIndChild = -1

                if self.interConc:
                    print("\tВыбираем ребро с минимальным весом:")

                 for indChild in range(len(self.graf[numVer].child)):
# Выбор первого ребра для перехода
                              if self.graf[numVer].child[indChild][2]:
# Если по ребру ещё не проходили
                        bufWeight = self.graf[numVer].child[indChild]
[1]
                        bufIndChild = indChild
```

```
                        if self.interConc:
                            print(f"\t\tТекущее ребро с минимальным
весом, по которому ещё не проходили: "
                                  f"{chr(numVer  +
ord('a'))}——{bufWeight}—>{chr(bufIndChild + ord('a') + 1)}")
                        break

                for indChild in range(len(self.graf[numVer].child)):
# Нахождение ребра с наименьшим весом
                    if  self.graf[numVer].child[indChild][1] <
bufWeight and self.graf[numVer].child[indChild][2]:
                        bufWeight = self.graf[numVer].child[indChild]
[1]
                        bufIndChild = indChild
                        if self.interConc:
                            print(f"\t\tНовое ребро с минимальным
весмо, по которому ещё не проходили: "
                                  f"{chr(numVer  +
ord('a'))}——{bufWeight}—>{chr(bufIndChild + ord('a') + 1)}")

            if bufIndChild >= 0:    # Если нашли ребро
                if self.interConc:
                    print(f"\t\tПереходим к новой вершине по
ребру: "
                          f"{chr(numVer  +
ord('a'))}——{bufWeight}—>{chr(bufIndChild + ord('a') + 1)}")
                self.graf[numVer].child[bufIndChild][2] = False
                numVer = self.graf[numVer].child[bufIndChild][0]
                self.result += chr(numVer + 97)
            else:                   # Если из текущей веришны нет
исходящих ребер
                if self.interConc:
                    print(f"\t\tИз вершины \"{chr(numVer +
ord('a'))}\" нет исходящих ребер, по которым ещё не "
                          f"проходили. ", end='')
                self.result = self.result[:-1]  # Убираем текущую
вершину из ответа
                if len(self.result) > 1:        # Получаем номер
предыдущей вершины
                    numVer = ord(self.result[-1]) - ord('a')
                else:
                    numVer = self.numStart
                if self.interConc:
                    print(f"Переходим к предыдущей
вершине \"{chr(numVer + ord('a'))}\"")
            if self.interConc:
                print('')

    def algAStar(self):     # Алгоритм A*
        if self.interConc:
            print("\033[32mЗапуск алгоритма A*:\033[0m")
        numVer = self.numStart

        queueVer = []
        passedVer = []
```

```
            g = [-1] * 26
            f = [-1] * 26

                queueVer.append(numVer)       # Добавляем в очередь
начальную вершину
               g[numVer] = 0                   # Т.к. расстояние из
начальной вершины в саму себя без прохода по другим вершинам 0
            if self.individual:
                f[numVer] = float(g[numVer] + self.graf[numVer].heur)
            else:
                                    f[numVer]   =   float(g[numVer]   +
abs(ord(self.graf[numVer].symbol)                                     -
ord(self.graf[self.numFinish].symbol)))

            if self.interConc:
                           print(f"\tДобавляем в очередь начальную
вершину \"{self.graf[numVer].symbol}\" с приоритетом {f[numVer]}")
                print(f"\tОчередь с приоритетом: ", end='')
                for e in queueVer:
                    print(f"{self.graf[e].symbol} [{f[e]}]\n")

             while len(queueVer):     # Пока не будут обработаны все
вершины графа
                current = self.getVer(queueVer, f)
                if self.interConc:
                    print(f"\tТекущая очередь: ", end='')
                    for e in queueVer:
                           print(f"{self.graf[e].symbol} [{f[e]}]  ",
end='')
                                    print(f"\n\tИзвлекаем  из  очереди
вершину \"{self.graf[current].symbol}\"")

                  if current == self.numFinish:    # Если достигнута
конечная вершина
                    if self.interConc:
                              print(f"\tИзвлеченная вершина является
конечной вершиной\n")
                    self.createResult()
                    return

                  queueVer.remove(current)     # Удаляем из очереди,
т.к. обработали
                   passedVer.append(current)    # Добавляем в список
обработанных вершин

                if self.interConc:
                           print(f"\tПроверяем всех детей выбранной
вершины:")

                 for child in self.graf[current].child:  # Перебираем
всех детей текущей вершины
                    tentativeScore = g[current] + child[1]  # Путь до
ребенка от начальной вершины
                    if self.interConc:
```

```
                                                print(f"\t\tДля
вершины \"{self.graf[child[0]].symbol}\":\n\t\t\tэвристическая функция
- ", end='')
                        if self.individual:
                            print(self.graf[child[0]].heur)
                        else:
                            print(f"{abs(ord(self.graf[child[0]].symb
ol) - ord(self.graf[self.numFinish].symbol))}")
                        print(f"\t\t\tпуть от начальной вершины -
{tentativeScore}")
                    # Если ребенка ещё не обрабатывали или новый путь
до него короче
                    if child[0] not in passedVer or tentativeScore <
g[child[0]]:
                        self.graf[child[0]].parent = current
                        g[child[0]] = tentativeScore
                        if self.individual:
                                    f[child[0]] = g[child[0]] +
self.graf[child[0]].heur
                        else:
                                    f[child[0]] = g[child[0]] +
abs(ord(self.graf[child[0]].symbol) -
                                                    ord(self.graf
[self.numFinish].symbol))
                        if child[0] not in queueVer:     # Если
ребенка ещё нет в очереди
                            if self.interConc:
                                                print(f"\t\tДобавляем
вершину \"{self.graf[child[0]].symbol}\" в очередь с приоритетом "
                                            f"{f[child[0]]}")
                            queueVer.append(child[0])
                self.result = ''
                if self.interConc:
                    print("")

        def getVer(self, queueVer, f):   # Получение элемента из
очереди с приоритетом
            bufI = -1
            bufF = -1
            for i in queueVer:  # Перебор элементов очереди
                  if (f[i] <= bufF or bufF < 0) and f[i] >= 0:     #
Найден меньший приоритет
                    bufI = i
                    bufF = f[i]
            return bufI

        def createResult(self):     # Создание результата при помощи
прохода по графу в обратном направлении
            if self.interConc:
                    print(f"\tСохранение пути при помощи прохода в
обратном направлении:")

            numVer = self.numFinish
             self.result += str(chr(numVer + ord('a')))  # Добавляем
конечную вершину
```

```
            if self.interConc:
                        print(f"\t\tДобавляем в результат конечную
вершину \"{self.graf[numVer].symbol}\"")
                print(f"\t\tТекущий результат: {self.result}\n")

            while True:     # Проходим по всем вершинам
                if self.interConc:
                    bufNum = numVer

                numVer = self.graf[numVer].parent
                     if numVer != None:   # Если текущая вершина не
начальная
                    self.result += chr(numVer + ord('a'))

                    if self.interConc:
                                print(f"\t\tПри работе алгоритма в
вершину \"{self.graf[bufNum].symbol}\" попали из вершины "
                                f"\"{self.graf[numVer].symbol}\"")
                               print(f"\t\tДобавим её в результат.\n\t\
tТекущий результат: {self.result}\n")
                else:   # Если текущая вершина начальная
                    if self.interConc:
                                print(f"\t\tПри работе алгоритма в
вершину \"{self.graf[bufNum].symbol}\" попали из начальной "
                                f"вершины\n")
                    break
               self.result = self.result[::-1]       # Переворачиваем
результат

            if self.interConc:
                print("\t\tПеревернем результат")
                print(f"\t\tПолучим: {self.result}")


     def firstTask(interConc):       # Первое задание
          start, finish = input().split()      # Считывание старта и
финиша

         edges = []
         while True:     # Считывание ребер
             try:
                 edge = input(' ')
                 if edge == '' or edge == " ":   # Если ввод закончен
                     break
             except (EOFError):  # Если ввод закончен
                 break
             edges.append(list(edge.split()))

         graf = Graf(start, finish, interConc, False)
         graf.createGraf(edges)
         graf.greedyAlg()
         graf.printResult()
```

```
    def secondTask(interConc):      # Второе задание
        start, finish = input().split()  # Считывание старта и финиша

        edges = []
        while True:  # Считывание ребер
            try:
                edge = input(' ')
                if edge == '' or edge == " ":   # Если ввод закончен
                    break
            except (EOFError):  # Если ввод закончен
                break
            edges.append(list(edge.split()))

        graf = Graf(start, finish, interConc, False)
        graf.createGraf(edges)
        graf.algAStar()
        graf.printResult()


    def  individualTask(interConc):             #  Второе  задание  с
индивидуализацией
        start, finish = input().split()  # Считывание старта и финиша

        edges = []
        while True:  # Считывание ребер
            try:
                edge = input(' ')
                if edge == '' or edge == " ":  # Если ввод закончен
                    break
            except (EOFError):  # Если ввод закончен
                break
            edges.append(list(edge.split()))

        graf = Graf(start, finish, interConc, True)
        graf.createGraf(edges)
        print("Введите значения эвристических функций:")
        for ver in graf.graf:
            if ver.symbol != None:
                print(f"{ver.symbol}: ", end='')
                ver.heur = int(input())
        print("")
        graf.algAStar()
        graf.printResult()


    if __name__ == "__main__":
        #firstTask(True)
        secondTask(False)
        #individualTask(True)
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.1 - Тестирование для 1 задания

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 10.0 f g 1.0	abdeag	
2.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde	
3.	a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	abcd	
4.	a b a b 1.0 a c 1.0	ab	
5.	a a a b 7 b c 1	a	

	b a 3 c a 0		
6.	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	bge	

Таблица Б.2 — Тестирование для 2 задания

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 10.0 f g 1.0	abefg	
2.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	
3.	a d	aed	

	a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0		
4.	a b a b 1.0 a c 1.0	ab	
5.	a a a b 7 b c 1 b a 3 c a 0	a	
6.	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	bge	

Таблица Б.3 — Тестирование индивидуализации

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0	abdefg	

	d e 4.0 e a 1.0 e f 2.0 a g 10.0 f g 1.0 1 2 3 4 5 6 7		
2.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 2 3 7 9 0	abcde	
3.	a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0 0 0 0	aed	

	0 0		
4.	a b a b 1.0 a c 1.0 1000 1 0	ab	
5.	a a a b 7 b c 1 b a 3 c a 0 7 9 8	a	
6.	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0 4 0 10 2 7 8	bge	